Kapittel 3

Instruksjoner

Innhold

3.1 Læringsmål 50
3.2 Fra språk til maskin 50
3.3 Rommet med hundre postkasser 51
3.4 Ordren som et tall 53
3.5 Hent og utfør 55
3.6 Instruksjonssettet og de første programmene 58
3.7 Assembler: navn i stedet for tall 60
3.8 Kontrollflyt 61
3.9 Adresser, etiketter og data 64
3.10 Oppsummering 68
3.11 Videre lesning 69
3.12 Refleksjon 69
3.13 Selv-test 70

Et program er en tekst du skriver. En prosessor er et stykke silisium. Mellom de to ligger dette kapitlet.

I 1930-årenes USA fantes det rom fulle av mennesker som regnet. De ble kalt *computers*, for det var det ordet betydde den gangen: en som regner. De fleste av dem var kvinner. Hver av dem hadde en mekanisk regnemaskin, en bunke papir og en liste med instruksjoner. Listen kunne si: ta tallet i rad 17, gang med 2,71828, rund av til fire desimaler, skriv svaret i rad 18.

Ingen av dem trengte å vite hva tabellen skulle brukes til. De fulgte listen. Det er nettopp derfor arbeidet lot seg overta av en maskin.

Tankevekker

Et program er tall i minnet. Maskinen kan ikke se forskjell på et tall den skal regne med og et tall den skal adlyde. Forskjellen ligger bare i hvordan tallet blir brukt.

I kapittel 2 så vi at et bitmønster ikke betyr noe uten en tolkning. Nå møter vi den viktigste tolkningen av dem alle. Maskinen leser et tall og gjør det tallet sier.

3.1 Læringsmål

Etter dette kapitlet skal du kunne

- **forklare** hva som skjer med en Python-linje på veien ned til maskinen
- **dele** en instruksjon i operasjonskode og adressedel
- **gjøre rede for** hent-utfør-syklusen og hva programtelleren peker på
- **spore** et LMC-program celle for celle og forutsi hva det skriver ut
- **skrive** små programmer med LMC sitt instruksjonssett
- **oversette** mellom maskinkode og assemblerkode
- **bruke** betingede hopp til å bygge løkker og valg
- **finne** feilen i en løkke som ikke terminerer
- **skille** en adresse fra verdien som ligger der
- **forklare** hvorfor et program kan endre seg selv, og hva det koster

3.2 Fra språk til maskin

Du har skrevet Python. Du har skrevet noe slikt som dette:

```
svar = a + b
```

Én linje. Men prosessoren i maskinen din kan ikke lese den linja. Den kan ikke lese bokstaver i det hele tatt.

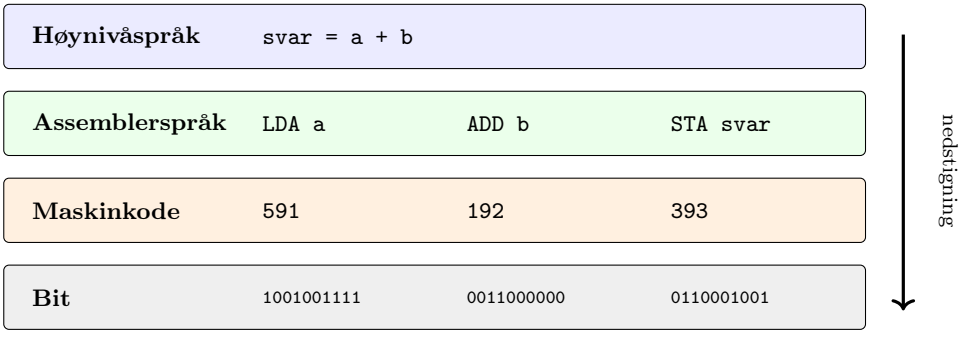
Så noe må skje på veien. Det som skjer, er en nedstigning gjennom flere språk.

3.2.1 Stigen

Øverst ligger *høynivåspråk* som Python, Java og C. De er laget for mennesker. Du skriver `a + b`, ikke hvor `a` ligger i minnet.

Nederst ligger *maskinkode*. Det er tall. Ingenting annet.

Mellom dem ligger *assemblerspråk*. Det er maskinkode med navn på. I stedet for tallet 5 skriver du `LDA`. Ett navn per instruksjon.



Figur 3.1: Den samme beregningen på fire nivåer. Hvert trinn ned fjerner et navn og legger til et tall. Nederste rad er de tre tallene skrevet binært.

3.2.2 Hvem gjør nedstigningen?

Det finnes to hovedmåter.

En *kompilator* oversetter hele programmet én gang, før det kjøres. Resultatet er en fil med maskinkode. C og Rust gjør det slik.

En *tolk* tar programmet linje for linje mens det kjører. Python gjør i hovedsak det. Kildekoden blir først oversatt til *bytekode*, en mellomform. Så er det et annet program, selve tolken, som leser bytekoden og gjør det den sier. Tolken er selv skrevet i C og finnes som maskinkode.

Det betyr at det aldri finnes maskinkode for *din* Python-linje. Det finnes maskinkode for tolken, og tolken adlyder linja di.

Note-to-myself

Forskjellen ligner den mellom å få en bok oversatt én gang for alle, og å ha en tolk ved siden av seg som oversetter setning for setning mens du leser. Den ferdige oversettelsen er raskere å lese. Tolken er raskere å komme i gang med.

Et *assemblerprogram* gjør den siste og enkleste oversettelsen. Det bytter `LDA` med 5 og et navn med en adresse. Ett navn inn, ett tall ut.

3.2.3 Hvorfor vi går helt ned

Det er ikke sikker du kommer til å skrive assemblerkode igjen, men nedstigningen er likevel verdt turen av flere grunner.

Den første er at abstraksjonene lekker. Når Python-programmet ditt er uforklarlig tregt, eller krasjer med en feilmelding om minne, ligger svaret under.

Den andre er at sikkerhetshull bor her nede. Et program som skriver til feil adresse, gjør nøyaktig det det fikk beskjed om. Skaden ligger i beskjeden, og beskjeden er et tall.

Den tredje er at hele resten av boka bygger på dette kapitlet. Kapittel 4 viser kretsene som utfører instruksjonene. Kapittel 5 viser portene de er bygd av.

Nederst på stigen ligger en liste med svært små ordrer. Hver ordre er et tall.

Fun Fact

Objektorientert programmering ble oppfunnet i Oslo. Ole-Johan Dahl og Kristen Nygaard utviklet Simula i 1960-årene. Klasser, objekter og arv kom derfra, og gikk videre til blant annet C++, Java og Python. Begge var professorer ved Universitetet i Oslo. Det er også derfor informatikk bygningene heter Ole Johan Dahls hus og Kristen Nygaards hus.

3.3 Rommet med hundre postkasser

Vi skal bruke en lekemaskin. Den heter *Little Man Computer*, forkortet LMC, og den ble laget av Stuart Madnick ved MIT. Simulatoren vi bruker er laget av Peter Higginson. Den er teoretisk korrekt og den er bare veldig liten. Hovedformålet er å få en forenklet inngang til hvordan prosessering i maskinvare og instruksjoner henger sammen.

Det som gjør LMC brukbar, er at den regner i titallssystemet. En ekte maskin gjør alt i binært, og da

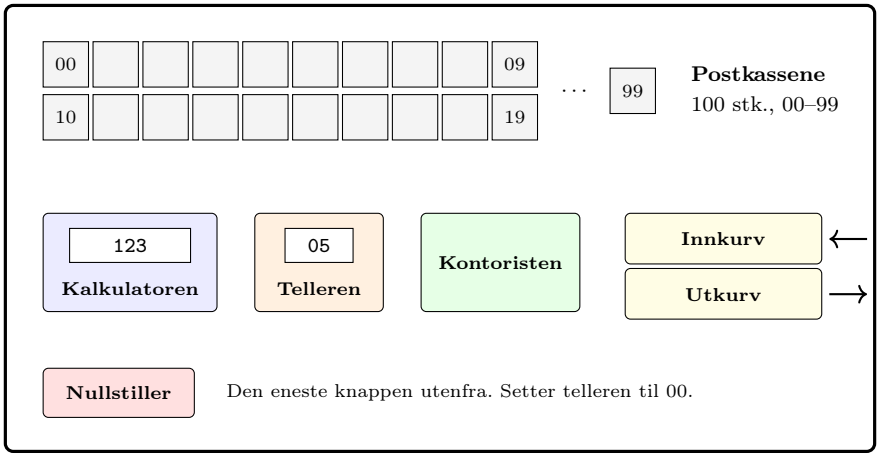
må du tenke på to ting samtidig: hvordan en instruksjon virker, og hvordan den er kodet i bit. LMC lar deg utsette det ene. Kodingen tar vi i kapittel 4.

Se for deg et lukket rom, som i figur 3.2. Langs veggen henger hundre postkasser. De er nummerert fra 00 til 99. Hver postkasse rommer én lapp, og på lappen står det et tresifret tall.

I rommet arbeider en kontorist. Kontoristen har fire ting til rådighet:

- **En kalkulator.** Den kan legge sammen og trekke fra. Displayet viser tre sifre.
- **En teller.** Den viser to sifre, og kontoristen kan klikke den ett hakk opp eller stille den direkte.
- **En innkurv.** Her legger noen utenfra lapper med tall.
- **En utkurv.** Her legger kontoristen lapper med tall.

Utenfor rommet sitter én knapp: en nullstiller som setter telleren til 00 og starter kontoristen på nytt. Bortsett fra den knappen og de to kurvene er rommet lukket.



Alt som går inn og ut av rommet, er tresifrede tall.

Figur 3.2: LMC-rommet. Kontoristen kan bare det som finnes i dette rommet.

3.3.1 Hvorfor akkurat hundre postkasser?

Fordi en adresse skal få plass i to sifre. To sifre gir tallene 00 til 99, altså hundre muligheter[1]. Flere postkasser ville krevd et tredje siffer i adressen, og da hadde det ikke vært plass til noe annet i en tresifret instruksjon.

Det er samme type regnestykke som i kapittel 2, der vi spurte hvor mange bit et tall trenger. Her er det sifre i stedet for bit, men avveiningen er den samme. Adressefeltet koster plass (bits) og antall plasser (bits) er alltid stor konkurranse om.

3.3.2 Innhold er ikke adresse!

Postkasse 42 er et sted. Tallet 42 er en verdi. De har ingenting med hverandre å gjøre.

[1] Merk at vi teller 0 med som det første tallet og da er 0-99 faktisk 100 tall :)

Tenk på en pakkeboks på et utleveringssted. Boksnummeret forteller hvor du skal gå. Det forteller ingenting om hva som ligger i boksen.

Postkasse 42 kan inneholde tallet `542`. Postkasse 542 finnes ikke. Og tallet 42 kan ligge i hvilken som helst av de hundre postkassene.

Advarsel :)

Når vi skriver `ADD 99`, er 99 en adresse. Instruksjonen betyr «legg til det som ligger i postkasse 99», ikke «legg til 99». LMC har ingen måte å si «legg til tallet 99» direkte. Skal du legge til 99, må tallet 99 først ligge i en postkasse.

3.3.3 Den regelen som bærer resten av kapitlet

Hver postkasse inneholder et tresifret tall. Ingenting ved postkassen forteller om tallet er noe å regne med eller en ordre å følge.

Kontoristen kan ikke se forskjell. Det finnes ingen forskjell å se.

Hva tallet er, avgjøres av én ting: om kontoristen kommer dit for å hente en ordre, eller for å hente en verdi. Det er hele forskjellen, og den ligger ikke i tallet.

Note-to-myself

Vi kommer tilbake til denne regelen fire ganger i dette kapitlet. Første gang som en advarsel om hvor data skal ligge. Andre gang som forklaringen på hvorfor et program uten stoppinstruksjon oppfører seg vilt. Tredje gang som et triks. Fjerde gang som et sikkerhetsproblem.

3.3.4 Simulatoren

LMC finnes som en simulator du kan kjøre i nettleseren, laget av Peter Higginson. Der ser du postkassene som et rutenett, og du kan taste inn tall direkte i cellene. Du kan kjøre programmet med `Run`, eller ett steg om gangen med `Step`.

Bruk `Step` i begynnelsen. Hele poenget med LMC er at du kan se hva som skjer.

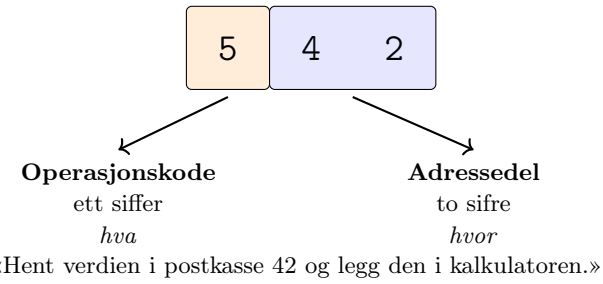
Advarsel :)

Husk `Reset` mellom to kjøringer. Glemmer du den, starter maskinen der forrige kjøring sluttet, og da stemmer ingenting. Dette er den vanligste kilden til forvirring de første timene med LMC.

3.4 Ordren som et tall

En instruksjon i LMC er et tresifret tall. Det er alt en instruksjon er.

Tallet leses i to deler, slik figur 3.3 on the following page viser. Det første sifferet sier *hva* som skal gjøres. De to siste sier *hvor*.



Figur 3.3: Instruksjonen `542`, delt i de to feltene sine.

Operasjonskoden kalles ofte bare *opkode*, fra engelsk *opcode*. Hvilket tall som betyr hva, er et valg noen har tatt. Det følger ikke av noe. Den som designer maskinen, bestemmer.

Note-to-myself

I LMC har vi ett siffer for *opkode* som betyr at vi kan bare gjøre 10 ulike operasjoner (instruksjoner). Hva de forskjellige tallene betyr kommer vi til i kapittel 3.6 on page 58.

3.4.1 Tre eksempler

`542` er opkode 5 og adresse 42. Hent verdien i postkasse 42.

`199` er opkode 1 og adresse 99. Legg verdien i postkasse 99 til det som står i kalkulatoren.

`000` er opkode 0. Stopp.

Legg merke til at `000` også har en adressedel. Den er bare null, og den brukes ikke. Stoppinstruksjonen bryr seg ikke om adressen.

Note-to-myself

Fordi adressedelen ignoreres, stopper *alle* tall som begynner på null. `017` stopper maskinen like effektivt som `000`. Vi skriver alltid `000`, men det er sifferet lengst til venstre som gjør jobben.

3.4.2 Adressen som ikke er en adresse

Noen instruksjoner trenger ikke en postkasse i det hele tatt, f.eks. å lese et tall fra innkurven. Det finnes ingen adresse å oppgi. Kurven er kurven!

Da står adressedelen ledig, og den blir brukt til noe annet: å skille mellom flere instruksjoner som deler samme opkode.

- `901` leser et tall fra innkurven.
- `902` skriver tallet i kalkulatoren til utkurven.
- `922` skriver ut ett tegn.

Alle tre har opkode 9. Tallene 01, 02 og 22 forlenger opkoden, og ingen av dem viser til en postkasse.

Advarsel :)

`901` betyr ikke «gjør noe med postkasse 01». Ingen postkasse er innom. Hvis du leser `901` som en vanlig instruksjon med adresse, blir alt annet i dette kapitlet vanskeligere enn det trenger å være.

I en ekte maskin brukes det samme trikset, og der er det mindre vilkårlig enn det ser ut. Adressefeltet i en inn- og ut-instruksjon inneholder gjerne nummeret til en enhet. Skjermen har ett nummer, disken et annet. Adressen peker fortsatt på noe, bare ikke på minnet.

3.4.3 Hvor mange ordrer finnes det?

LMC bruker ni opkoder: 0, 1, 2, 3, 5, 6, 7, 8 og 9. Opkode 4 står ledig i denne utgaven av maskinen.

Instruksjonene er litt flere enn opkodene, fordi opkode 9 dekker tre stykker. Til sammen blir det elleve.

Til sammenligning har prosessoren i maskinen din flere tusen instruksjoner. Elleve er lite, men elleve er nok. Vi kommer til å regne, velge og gjenta med dem alle sammen.

Tankevekker

Kraften i en datamaskin kommer ikke av at operasjonene er avanserte. Den kommer av at svært enkle operasjoner utføres svært mange ganger i sekundet. Alt det kompliserte ligger i rekkefølgen.

3.4.4 Det samme, i bit

LMC teller i titallssystemet. En ekte maskin gjør ikke det.

I en ekte maskin er instruksjonen et bitmønster, og de to feltene er grupper av bit i stedet for sifre. Avveiningen blir den samme, men vesentlig skarpere. Hvor mange bit skal opkoden ha, og hvor mange skal adressen ha? Flere bit til opkoden gir flere instruksjoner. Flere bit til adressen gir mer minne å peke på.

Den regningen tar vi i kapittel 4. Her holder det å vite at delingen i to felt er den samme, uansett tallsystem.

3.5 Hent og utfør

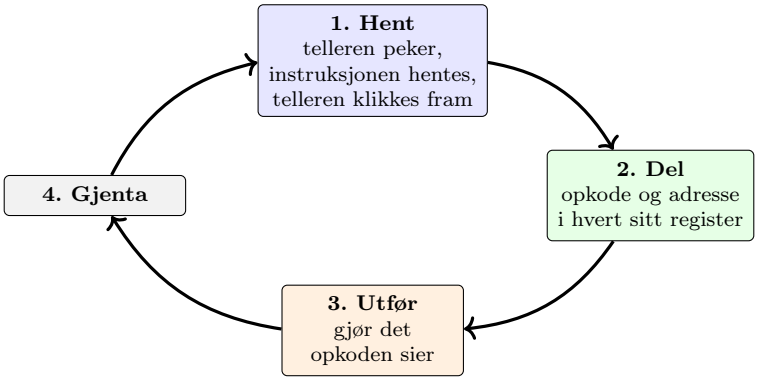
Nå vet vi hva en ordre er. Nå skal vi se hvordan den blir utført.

Kontoristen gjør det samme om og om igjen, i en løkke som aldri varierer. Den løkka heter *hent-utfør-syklusen*.

3.5.1 De fire stegene

1. Bruk verdien i telleren som adresse, og hent instruksjonen som ligger i den postkassen. Klikk samtidig telleren ett hakk fram.
2. Del instruksjonen i to. Opkoden går i instruksjonsregisteret, adressedelen i adresseregisteret.
3. Gjør det instruksjonsregisteret sier.
4. Begynn på nytt fra steg 1.

Det er hele maskinen, og figur 3.4 viser runden. Maskinen gjentar den til den treffer en stoppinstruksjon.



Figur 3.4: Hent-utfør-syklusen. Den stopper bare når opkoden er 0.

3.5.2 De fire registrene

Steg 2 nevnte to registre. Et *register* er en liten lagringsplass inne i prosessoren, langt raskere å nå enn en postkasse.

LMC har fire.

- **Akkumulatoren** er kalkulatoren. Her ligger tallet det regnes med, og hit kommer svaret.
- **Programtelleren** er telleren. Den sier hvilken instruksjon som skal hentes *neste* gang.
- **Instruksjonsregisteret** holder koden til instruksjonen som utføres nå.
- **Adresseregisteret** holder adressedelen av den samme instruksjonen.

De to siste finnes ikke som gjenstander i rommet. Ordren må huskes underveis, fra den hentes til den er utført, og det er den huskingen de to registrene står for.

En moderne prosessor har fra tjue til et par hundre registre som programmet kan se, og noen flere typer enn disse. Den deler dessuten opp arbeidet litt annerledes: der vi har ett register til koden og ett til adressen, har en ekte maskin egne registre for å snakke med minnet. Kapittel 4 tar dem.

3.5.3 Når klikkes telleren?

Dette er det punktet der flest går seg vill, så vi tar det sakte.

Telleren klikkes fram i steg 1, samtidig med hentingen. Ikke etterpå.

Grunnen er en definisjon som lønner seg å holde fast ved:

Tankevekker

Programtelleren peker aldri på instruksjonen som utføres nå. Den peker på den neste.

Tenk på en nummerert huskeliste. Du holder plassen med fingeren. Fornuftig rekkefølge er å flytte fingeren ned til neste linje *før* du gjør det linja sa. Ellers mister du plassen din i det øyeblikket oppgaven sender deg et annet sted.

Og det er akkurat det et hopp gjør. Et hopp sender deg et annet sted.

3.5.4 Derfor virker telleren etter hopp

Sett at instruksjonen i postkasse 07 sier «hopp til 01».

Kontoristen henter den fra 07 og klikker telleren til 08. Så utføres ordren, og ordren sier: sett telleren til 01. Telleren blir 01.

Klikket var ikke bortkastet. Det ble bare overskrevet.

Hadde telleren blitt klikket *etter* at ordren var utført, ville hoppet blitt ødelagt. Telleren hadde blitt satt til 01, og så klikket til 02. Hoppet hadde bommet med én.

Note-to-myself

Rekkefølgen på stegene inne i én instruksjon betyr like mye som rekkefølgen på instruksjonene i programmet. Det er lett å glemme, fordi den første rekkefølgen er usynlig.

3.5.5 Én runde, steg for steg

La oss følge én instruksjon helt gjennom. Utgangspunktet er dette:

- Telleren står på 25.
- Postkasse 25 inneholder `589`.
- Postkasse 89 inneholder `222`.
- Kalkulatoren viser `333`.

Steg	Hva som skjer	Etterpå
1	Les telleren, hent fra postkasse 25	Instruksjonen er `589`, telleren er 26
2	Del `589` i 5 og 89	Opkode 5, adresse 89
3	Opkode 5 betyr hent: les postkasse 89	Kalkulatoren viser `222`
4	Begynn på nytt	Neste instruksjon hentes fra 26

Tabell 3.1: Én runde i syklusen. Postkasse 89 er uendret etter steg 3, og innholdet i kalkulatoren er borte.

To ting å merke seg. Postkasse 89 er uendret, for kontoristen leste bare lappen og la den tilbake. Og tallet 333 er borte for godt, for kalkulatoren har bare plass til ett tall.

3.5.6 Den viktigste konsekvensen

Syklusen spør aldri om det den hentet er en instruksjon.

Steg 1 henter et tall. Steg 3 gjør det tallet sier. Det finnes ikke noe steg som sjekker om tallet var ment som en ordre.

Legger du data der kontoristen kommer til å hente sin neste ordre, blir dataene utført. Maskinen protesterer ikke. Dette er viktig å bemerke seg da det ofte er her både manipulering (hacking) kan skje og at maskinen utfører noe den egentlig ikke skulle ha utført.

3.6 Instruksjonssettet og de første programmene

Her er alle ordrene kontoristen forstår.

Kode	Navn	Hva den gjør
0xx	HLT	Stopper kjøringen
1xx	ADD	Legger verdien i postkasse xx til akkumulatoren
2xx	SUB	Trekker verdien i postkasse xx fra akkumulatoren
3xx	STA	Lagrer akkumulatoren i postkasse xx
4xx	–	Ikke i bruk
5xx	LDA	Henter verdien i postkasse xx til akkumulatoren
6xx	BRA	Hopper til adresse xx
7xx	BRZ	Hopper til xx hvis akkumulatoren er 0
8xx	BRP	Hopper til xx hvis akkumulatoren er 0 eller større
901	INP	Leser et tall fra innkurven til akkumulatoren
902	OUT	Skriver akkumulatoren til utkurven
922	OTC	Skriver akkumulatoren som ett ASCII-tegn
–	DAT	Setter av en postkasse til data

Tabell 3.2: Hele instruksjonssettet til LMC. De fire siste radene har ingen adressedel å fylle ut.

De korte navnene kalles *mnemonics*, av et gresk ord for hukommelse. Vi kommer tilbake til dem i seksjon 3.7. Foreløpig er de bare en lesehjelp.

3.6.1 Det første programmet

Vi skal legge sammen to tall. Brukeren legger to lapper i innkurven, og summen kommer ut.

```
Postkasse   Innhold
00          901       les første tall
01          399       lagre det i postkasse 99
02          901       les andre tall
03          199       legg til det som ligger i 99
04          902       skriv ut svaret
05          000       stopp
...
99          000       her ligger dataene
```

Hvorfor må det første tallet lagres i det hele tatt[2]? Fordi kalkulatoren bare har plass til ett tall. Leser vi det andre tallet rett inn, blir det første borte.

Det er hele grunnen til at `STA` finnes. Akkumulatoren er en arbeidsbenk, ikke et lager.

[2]Vi har i dette eksemplet valgt postkasse 99, men kunne i prinsippet valgt hvilke som helst annet. Mer om det i kapittel 3.6.2 on the next page

Postkasse	Instruksjon	Akkumulator	Postkasse 99	Hva skjer
00	`901`	12	`000`	Brukeren taster 12
01	`399`	12	`012`	12 kopieres til 99
02	`901`	5	`012`	Brukeren taster 5
03	`199`	17	`012`	$5 + 12 = 17$
04	`902`	17	`012`	17 havner i utkurven
05	`000`	17	`012`	Kontoristen tar pause

Tabell 3.3: Addisjonsprogrammet kjørt med tallene 12 og 5. Første kolonne er postkassen instruksjonen ble hentet fra. Kolonnene «Akkumulator» og «Postkasse 99» viser verdiene *etter* at instruksjonen er utført. Telleren står allerede ett hakk lenger fram hele veien.

Legg merke til rad 4. Postkasse 99 inneholder fortsatt 012 etter at summen er regnet ut. `ADD` leser postkassen, den tømmer den ikke.

3.6.2 Hvorfor postkasse 99?

Fordi den er langt unna programmet.

Programmet opptar postkassene 00 til 05. Enhver postkasse etter 05 hadde fungert like godt. Vi valgte 99 fordi det er tydelig utenfor. Kan du kanskje tenke deg hvordan man burde overveie bruken av postkassene? Hvordan vite hvor mange postkasser programmet trenger for sine instruksjoner og hvor langt unna du har til å lagre ting.

Legger vi derimot dataene i postkasse 03, skjer det noe stygt. Instruksjonen som lå der, blir borte. Og når kontoristen kommer til 03 for å hente sin neste ordre, ligger det et datatall der i stedet.

Det tallet blir utført. Hva maskinen da gjør, kan ingen forutsi uten å regne på det.

Advarsel :)

Å glemme `HLT` på slutten er den vanligste feilen i LMC. Uten stoppinstruksjon fortsetter kontoristen inn i dataene og utfører dem. Hva som da skjer, avhenger av hvilke tall dataene tilfeldigvis er. Vi ser et konkret eksempel på neste side.

3.6.3 Tre tall i stedet for to

Samme oppgave, men nå skal tre tall legges sammen.

```
00  901     les første tall
01  399     lagre i 99
02  901     les andre tall
03  199     legg til
04  399     lagre summen i 99 igjen
05  901     les tredje tall
06  199     legg til
07  902     skriv ut
08  000     stopp
99  000     data
```

Mønsteret er: les, legg til, lagre tilbake. Det er det samme mønsteret som en løkke bruker, og vi møter det igjen i seksjon 3.8.

3.6.4 Et program som skriver tekst

I kapittel 2 lærte vi at tegn er tall. Bokstaven `h` er 104 i ASCII, `e` er 101 og `i` er 105.

Instruksjonen `922` tar tallet i kalkulatoren og skriver det ut som tegn. Da kan vi skrive et ord.

```
00  507       hent tallet i postkasse 07
01  922       skriv det ut som tegn
02  508       hent tallet i postkasse 08
03  922       skriv det ut som tegn
04  509       hent tallet i postkasse 09
05  922       skriv det ut som tegn
06  000       stopp
07  104       'h'
08  101       'e'
09  105       'i'
```

Programmet skriver `hei`.

Her er dataene lagt rett etter programmet, i postkassene 07, 08 og 09. Det er trygt, for kontoristen stopper på 06 og kommer aldri dit.

Tenk om...

Hva hadde skjedd hvis vi hadde glemt `HLT`, og satt en fjerde utskrift i postkasse 06 i stedet? Kontoristen ville kommet til postkasse 07 og hentet `104` som sin neste ordre. Opkode 1 er `ADD`, adressen er 04. Maskinen hadde lagt innholdet i postkasse 04 til kalkulatoren, som er `509`, og fortsatt videre til postkasse 08. Tallet 104 er både bokstaven `h` og ordren «legg til det som ligger i 04». Ingenting i tallet avgjør hvilken av delene det er.

3.7 Assembler: navn i stedet for tall

Se på dette programmet en gang til:

```
901  399  901  199  902  000
```

Hva gjør det?

Du vet svaret, for du leste det for to sider siden. Men du ser det ikke. Du husker det.

Det er problemet med maskinkode. Den er ikke laget for mennesker.

3.7.1 Fire grunner til at tall er tungvint

1. Du må huske hvilket tall som betyr hva.
2. Du kan ikke se hva et program gjør ved å se på det.

3. Du kan ikke se hvilke postkasser som er data og hvilke som er kode.
4. Setter du inn én ny instruksjon, forskyves alt under, og alle adresser må rettes.

Den fjerde er verst, og vi kommer tilbake til den i seksjon 3.9.

3.7.2 Løsningen

Vi gir instruksjonene navn. Så skriver vi et program som bytter navnene tilbake til tall for oss.

Det programmet heter en *assembler*, og språket heter assemblerspråk. Programmet over ser slik ut:

```
INP          les et tall
STA 99       og legg det i postkasse 99
INP          les et tall til
ADD 99       og legg sammen
OUT          skriv ut svaret
HLT          stopp
```

Nå ser du hva programmet gjør.

3.7.3 Hva assembleren faktisk er

En assembler er en én-til-én-oversettelse. Hver linje du skriver blir nøyaktig én postkasse i minnet. Ingen linjer legges til, ingen fjernes.

Det skiller assembler fra en kompilator. En kompilator kan gjøre om én linje Python til hundre maskininstruksjoner, eller til null hvis linja var overflødig. En assembler gjør ingen slike valg.

Tankevekker

Maskinen kjenner ikke `LDA`. Den har aldri sett de bokstavene. `LDA` er noe vi skriver, og assembleren er programmet som fjerner det igjen før maskinen får se noe.

Dette er nøyaktig den typen abstraksjon kapittel 1 handlet om. Grensesnittet er navnene. Implementasjonen er tallene. Og her kan du se gjennom abstraksjonen, for begge sider av den får plass på én side.

Resten av kapitlet bruker assembler. Det gjør programmene lesbare. Husk likevel hva som faktisk ligger i postkassene. Det ligger tall der. Det ligger alltid bare tall der.

3.8 Kontrollflyt

Alt vi har skrevet så langt, kjører rett fram. Første instruksjon, andre instruksjon, og videre nedover til stopp.

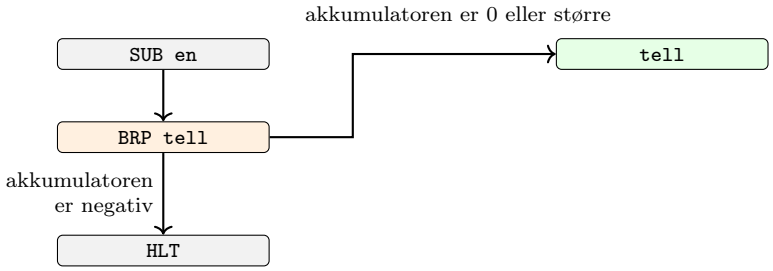
En maskin som bare kan det, er en kalkulator. Den kan gjøre én ting med tallene du gir den.

Det som gjør en datamaskin til noe annet, er at den kan velge. Og valget er bygd av én ting: muligheten til å sette programtelleren til hva som helst.

3.8.1 De tre hoppene

- `BRA xx` setter telleren til `xx`. Alltid.
- `BRZ xx` setter telleren til `xx` hvis akkumulatoren er null. Ellers skjer ingenting.
- `BRP xx` setter telleren til `xx` hvis akkumulatoren er null eller positiv. Ellers skjer ingenting.

Det er alt. Tre instruksjoner, og med dem kan du bygge enhver løkke og enhver betingelse som finnes i Python. Figur 3.5 viser hvordan et betinget hopp deler programmet i to veier.



Et betinget hopp er et veiskille. Maskinen tar den ene veien eller den andre, og valget avhenger *bare* av tallet i akkumulatoren.

Figur 3.5: Et betinget hopp. Instruksjonen som testes, står ikke i hoppet – den står over det.

Advarsel :)

Null er positivt i `BRP`. Instruksjonen hopper hvis akkumulatoren er null eller større, og faller bare gjennom hvis tallet er negativt. Dette er en klassisk felle, og den er årsaken til at nedtellingsprogrammet nedenfor stopper på -1 og ikke på 0.

3.8.2 Det hoppet ser på

`BRZ` og `BRP` ser på akkumulatoren. De ser ikke på instruksjonen over. De ser ikke på noen postkasse. De ser på ett tall, og det tallet ble satt av en tidligere instruksjon.

Det betyr at testen og det som testes, står to forskjellige steder i programmet. Mellom dem kan det stå hva som helst. Skriver du en `LDA` mellom dem, tester du på noe helt annet enn du trodde.

Note-to-myself

Vil du teste om et tall er 7, finnes det ingen instruksjon for det. Du må trekke fra 7 og teste mot null. Alle sammenligninger i LMC blir til en subtraksjon etterfulgt av et hopp. Slik er det i ekte maskiner også, og det er derfor de har et eget nullflagg.

3.8.3 En løkke som ikke stopper

Vi skal telle ned fra 10 og skrive ut hvert tall. Her er et forsøk.

```
00  LDA 05       hent 10
01  OUT          skriv ut
```

```
02  SUB 06       trekk fra 1
03  BRA 01       hopp tilbake
04  HLT
05  10           data: startverdien
06  1            data: tallet én
```

Kjør det, og maskinen skriver 10, 9, 8 og videre nedover. Den skriver også -1, -2, -3. Den kommer aldri til postkasse 04.

Feilen er `BRA`. Et ubetinget hopp hopper alltid.

Note-to-myself

I simulatoren stopper løkka likevel til slutt, men ikke fordi programmet er ferdig. Kalkulatoren har bare plass til tre sifre, så et sted under -999 går den tom og gir feilmelding. Det er en annen feil enn den vi leter etter.

3.8.4 Løkka som stopper

Bytt `BRA` med `BRP`:

```
00  LDA 05       hent 10
01  OUT          skriv ut
02  SUB 06       trekk fra 1
03  BRP 01       hopp tilbake hvis 0 eller større
04  HLT
05  10
06  1
```

Runde	Skrevet ut	Etter `SUB`	`BRP` gjør
1	10	9	hopper
2	9	8	hopper
⋮	⋮	⋮	⋮
10	1	0	hopper, for null er positivt
11	0	-1	hopper ikke

Tabell 3.4: Nedtellingen. Runde 11 skriver ut null og faller så gjennom til `HLT`.

Programmet skriver ut tallene fra 10 til 0, og stopper.

Tenk om...

Hva om vi heller ville stoppe *før* null ble skrevet ut? Da kunne vi lagt en `BRZ` et sted. Men hvor? Legger du den etter `SUB`, hopper programmet ut i det akkumulatoren når null, og null blir aldri skrevet ut. Legger du den før `OUT`, må du ha lastet verdien først. Rekkefølgen mellom test og utskrift avgjør hva som kommer med. Prøv begge i simulatoren.

3.8.5 Det samme i Python

Nedtellingen svarer til dette:

```
tall = 10
while tall >= 0:
    print(tall)
    tall = tall - 1
```

Fire linjer Python. Sju postkasser i LMC.

Forskjellen er ikke bare lengde. I Python står betingelsen øverst i løkka, og maskineriet som hopper er skjult. I LMC står hoppet nederst, og betingelsen er en subtraksjon lenger opp. Rekkefølgen snur.

Tankevekker

Byggeklossene i LMC er mye mindre enn dem du er vant til. Én `while`-linje er tre instruksjoner. Én `if` er to. Prosessoren i maskinen din bygger på nøyaktig samme vis, bare raskere og med flere klosser.

3.8.6 Et valg, ikke en løkke

Nå et program med `if`. Vi leser to tall og skriver ut forskjellen mellom dem, alltid som et positivt tall.

```
00  INP          les første tall
01  STA 10       lagre det
02  INP          les andre tall
03  STA 11       lagre det
04  SUB 10       andre minus første
05  BRP 08       hopp hvis svaret er positivt
06  LDA 10       ellers: snu regnestykket
07  SUB 11       første minus andre
08  OUT          skriv ut
09  HLT
10  0            data
11  0            data
```

Dette er en `if-else`. Linje 05 er testen. Linje 06 og 07 er `else`-grenen. Er svaret allerede positivt, hopper vi rett forbi dem.

Studer det til du kan følge begge veier. Dette mønsteret er alt en maskin trenger for å ta valg.

Note-to-myself

Dataene ligger i postkassene 10 og 11, rett etter `HLT`. Fjern `HLT` i postkasse 09, og kontoristen går rett inn i dataene. Postkasse 10 inneholder det første tallet brukeren tastet inn, så hva maskinen gjør, avgjøres av brukeren. Taster brukeren 7, blir tallet `007` lest som en stoppinstruksjon, og alt ser bra ut. Taster brukeren 512, blir det lest som `LDA 12`, og maskinen fortsetter inn i noe helt annet. Feilen viser seg altså bare for noen inndata, og det er den verste typen feil som finnes.

3.9 Adresser, etiketter og data

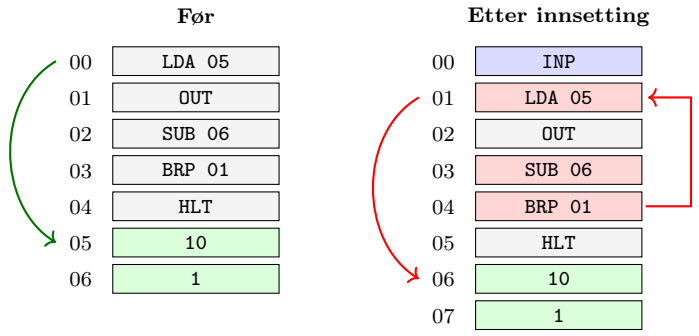
Programmene våre begynner å bli lange nok til å bli upraktiske. Og de er skjøre på en måte som er verdt å se på.

3.9.1 Problemet med tall som adresser

Se på nedtellingsprogrammet igjen. Startverdien ligger i postkasse 05.

Sett at du vil legge til en instruksjon på toppen, for eksempel en `INP` som lar brukeren velge hvor det skal telles fra. Alt under forskyves ett hakk. Dataene som lå i 05 og 06 ligger nå i 06 og 07.

Figur 3.6 viser hva som skjer. Instruksjonene sier fortsatt `LDA 05` og `SUB 06`. De peker feil. Og hoppet sier fortsatt `BRP 01`, som nå treffer `LDA` i stedet for `OUT`. Tre adresser er ødelagt, og ingenting sier fra.



Figur 3.6: Hvorfor talladresser ikke skalerer.

3.9.2 Etiketter

Løsningen er å gi postkassene navn.

En *etikett* er et navn på en adresse. Du skriver navnet i venstre kolonne for å døpe en postkasse. Du skriver det samme navnet i høyre kolonne for å peke på den.

Nedtellingen med etiketter:

```
        LDA start
tell    OUT
        SUB en
        BRP tell
        HLT
start   DAT 10
en      DAT 1
```

Nå kan du sette inn så mange instruksjoner du vil. Assembleren regner ut adressene på nytt hver gang.

Note-to-myself

Navnet betyr ingenting for maskinen. Du kan kalle postkassen `start`, `Amira` eller `x` uten at noe endrer seg. Etiketten forsvinner i det assembleren har gjort jobben sin, og det som ligger i minnet er et tall.

Er en etikett det samme som en variabel i Python? Nesten, men ikke helt. En variabel i Python er et navn på en verdi, og verdien kan flytte seg. En etikett er et navn på en *adresse*, og adressen ligger fast. Det er stedet som har navn, ikke innholdet.

3.9.3 DAT

`DAT` er ikke en instruksjon. Maskinen har ingen opkode for den.

`DAT` er en beskjed til assembleren: sett av en postkasse her, og legg eventuelt et starttall i den. Slike beskjeder kalles *pseudoinstruksjoner*.

Du kan skrive `DAT` uten tall. Da settes postkassen av uten at innholdet er bestemt. Det er sjelden lurt.

Advarsel :)

En `DAT` uten starttall inneholder det som lå der fra forrige kjøring. Programmet virker da kanskje første gang og feiler andre gang. Initialiser alltid dataene dine. Dette rådet gjelder i alle programmeringsspråk, men her ser du hvorfor.

3.9.4 Hva ligger egentlig i postkassen?

Tre spørsmål. Svar på hvert av dem før du leser videre.

```
Postkasse 00:   en      DAT 1
Postkasse 01:   peker   DAT en
Postkasse 02:   hent    LDA en
```

Hvilket *tall* står i hver av de tre postkassene?

I postkasse 00 står `001`. Det er verdien vi ba om.

I postkasse 01 står `000`. `en` er et navn på postkasse 00, og adressen til postkasse 00 er tallet 0.

I postkasse 02 står `500`. Opkoden for `LDA` er 5, og adressen til `en` er 00.

Tankevekker

En etikett er bare et tall. En instruksjon er bare et tall. Forskjellen mellom `peker DAT en` og `hent LDA en` er sifferet lengst til venstre, og ingenting annet.

Har du dette, har du hele resten av kapitlet.

3.9.5 Problemet med lister

Ta for deg at vi vil skrive ut en hel tekst. Teksten ligger i påfølgende postkasser.

Sett at teksten begynner i postkasse 12. Første bokstav hentes da med `LDA 12`, andre med `LDA 13`, tredje med `LDA 14` osv...

Instruksjonen må altså endre seg mellom hver runde i løkka. Men instruksjonene ligger fast i minnet.

Eller gjør de det?

Instruksjonen `LDA 12` er tallet `512`. Det ligger i en postkasse. Og et tall i en postkasse kan vi legge én til.

`512` pluss 1 er `513`, som er `LDA 13`. Legger vi til én på *instruksjonen*, peker den på neste postkasse.

```
Adr
00          LDA laster   hent tallet 512
```

```
01          STA hent     skriv det inn som instruksjon
02  hent    DAT 0        denne cellen overskrives hver runde
03          BRZ slutt    null betyr slutt på teksten
04          OTC          skriv ut tegnet
05          LDA hent     hent instruksjonen som et tall
06          ADD en       legg til én
07          STA hent     skriv den tilbake
08          BRA hent     kjør den på nytt
09  slutt   HLT
10  laster  DAT 512      dette er LDA 12, som et tall
11  en      DAT 1
12  tekst   DAT 104      'h'
13          DAT 101      'e'
14          DAT 105      'i'
15          DAT 0        null markerer slutten
```

Postkassen med etiketten `hent` inneholder en instruksjon som programmet selv skriver om mellom hver runde. Først `LDA 12`, så `LDA 13`, så `LDA 14`. Til slutt `LDA 15`, som henter nullen, og da hopper `BRZ` ut.

Legg merke til at `BRA hent` hopper til postkasse 02, ikke til 00. Runde to og utover starter altså midt i programmet, rett på den instruksjonen som nettopp ble skrevet om.

Og legg merke til én ting til: `laster DAT 512` er en adresse skrevet inn for hånd, i et kapittel som nettopp har forklart hvorfor det er en dårlig idé. Flytter du teksten, må du rette 512 selv. Slik er det med selvmodifiserende kode. Den bryter ut av det assembleren kan hjelpe deg med.

Advarsel :)

Det som legges til én, er *adressedelen* i instruksjonen. Ikke verdien som ligger på den adressen. Går du fra `512` til `513`, har du flyttet deg én postkasse bortover. Bokstaven du henter blir en annen, men ikke fordi bokstavtallet ble større. Dette er det spørsmålet studenter oftest stiller om selvmodifiserende kode.

3.9.6 Hva ekte maskiner gjør i stedet

En ekte prosessor har *indeksregistre*. Instruksjonen sier «hent fra adressen som står i registeret», og så teller du opp registeret i stedet for instruksjonen. Programmet ligger i ro.

Det kalles *indirekte adressering*, fordi adressen kommer et annet sted fra enn instruksjonen selv. LMC har det ikke. Kapittel 4 viser hvordan det er bygget.

3.9.7 Så er det lov, eller ikke?

Både og, og svaret er mer interessant enn det høres ut.

Moderne operativsystemer merker minnet ditt. En side kan være skrivbar, eller den kan være kjørbar. Som hovedregel ikke begge deler samtidig. Det gjør at et program ikke uforvarende, og ikke ondsinnet, kan skrive nye instruksjoner til seg selv.

Samtidig gjør noen programmer nettopp det, med vilje og med tillatelse. Nettleseren din oversetter JavaScript til maskinkode mens siden kjører, skriver koden i minnet og hopper til den. Det heter *just-in-time-kompilering*. Java og C# gjør det samme.

Og enda viktigere: for operativsystemet og for kompilatoren *er* programmet ditt data. Det må det være. Noen må lese det, flytte det og legge det i minnet før det kan bli et program.

Tankevekker

Grensen mellom kode og data er ikke noe maskinen finner. Den er noe vi håndhever, med vilje, fordi det ellers går galt. Den finnes i operativsystemet, ikke i silisiumet (maskinvaren).

3.9.8 Når det går galt

Et program som skriver til feil adresse, overskriver noe. Hva det overskriver, avhenger av hva som tilfeldigvis lå der.

Er det data, får du feil svar. Er det en instruksjon, kjører maskinen noe annet enn programmet ditt. Og er den feilaktige adressen noe en utenforstående kan påvirke, kan den utenforstående bestemme hva som blir kjørt.

Det er dette som ligger under en stor andel av sikkerhetshullene du leste om i første halvdel av emnet. Ikke et magisk angrep. Bare en maskin som utfører tallene den finner, slik den alltid har gjort.

Advarsel :)

Ikke bruk selvmodifiserende kode i vanlige programmer. Den er nesten umulig å feilsøke, den ødelegger for prosessorens forhåndshenting av instruksjoner, og den er blokkert på de fleste systemer. Her er den med fordi den avslører hva en maskin er, ikke fordi den er en god teknikk.

3.10 Oppsummering

Kapitlet startet med en Python-linje og endte med et program som redigerer seg selv. Veien mellom dem gikk gjennom én idé, gjentatt fire ganger.

Begrep	**Kort sagt**
Maskinkode	Programmet som tall. Det eneste maskinen faktisk har.
Assembler	Navn på tallene, og et program som fjerner navnene igjen.
Opkode	Første siffer. Sier hva som skal gjøres.
Adressedel	De to siste sifrene. Sier hvor, når det er aktuelt.
Akkumulator	Arbeidsbenken. Plass til ett tall om gangen.
Programteller	Adressen til den *neste* instruksjonen.
Hent-utfør	Hent, del, utfør, gjenta. Telleren klikkes under hentingen.
Betinget hopp	Setter telleren hvis akkumulatoren oppfyller en betingelse.
Etikett	Et navn på en adresse. Forsvinner når assembleren er ferdig.
`DAT`	Beskjed til assembleren om å sette av en postkasse.

Tabell 3.5: Begrepene fra dette kapitlet.

Den gjennomgående ideen kom igjen som en advarsel om hvor data skal ligge, som forklaringen på hva et program uten `HLT` gjør, som trikset bak selvmodifiserende kode, og til slutt som årsaken bak en hel klasse sikkerhetshull.

3.10.1 Videre til neste kapittel

Kontoristen var aldri der.

Vi har brukt hele kapitlet på en person i et rom, og det har vært nyttig, fordi du kan forestille deg selv i den rollen. Du kan flytte lapper mellom postkasser uten å vite hva regnestykket gjelder. Det er nettopp det en prosessor gjør.

Men det sitter ingen i maskinen din. Noe annet klikker telleren, deler instruksjonen i to og legger tall sammen. Det er kretser.

I kapittel 4 åpner vi rommet og ser hva som faktisk gjør jobben. Der blir også instruksjonen noe annet enn tre sifre. `542` blir et bitmønster, feltene blir grupper av bit, og hvor mange bit hvert felt skal ha blir et valg med konsekvenser.

Note-to-myself

Den samme ordren, i tre former. Som assembler: `LDA 42`. Som tall: `542`. Som bit, slik en ekte maskin lagrer den, er det bare et mønster av nuller og enere – og hvilket mønster, avhenger av hvordan maskinen er bygget. Det er neste kapittel.

3.11 Videre lesning

Little Man Computer. Englander kapittel 6 er den opprinnelige framstillingen, og den er kort og god. Merk at boka bruker `STO`, `IN` og `BR` der vi har brukt `STA`, `INP` og `BRA`.

Hent-utfør med registre. Englander 7.4 viser den samme syklusen med registeroverføringer skrevet ut, og med to registre vi ikke har hatt bruk for her. Les den når du har lest kapittel 4.

Hva slags instruksjoner finnes. Englander 7.5 går gjennom kategoriene i et ekte instruksjonssett: dataflytting, aritmetikk, logikk, skift, hopp og kall. LMC har de tre første i minimal utgave.

Simulatoren. Peter Higginsons LMC-simulator kjører i nettleseren. Skriv programmene i en tekstfil og lim dem inn, i stedet for å taste dem i nettleservinduet. Da mister du dem ikke.

På egen hånd. Finn ut hva prosessoren i din egen maskin heter, og søk opp instruksjonssettet. Du kommer til å kjenne igjen mer enn du tror.

3.12 Refleksjon

- **Om nedstigningen.** Python-linja `svar = a + b` ble til tre LMC-instruksjoner i figur 3.1 on page 50. Hva er det Python gjør for deg som LMC ikke gjør? Nevn minst tre ting, og avgjør for hver av dem om det koster tid under kjøring.
- **Om telleren.** Hvorfor definerer vi programtelleren som adressen til den *neste* instruksjonen, og ikke den som utføres nå? Hva hadde blitt vanskeligere med den andre definisjonen?
- **Om elleve instruksjoner.** LMC har elleve. Din egen prosessor har flere tusen. Hva bruker den de resterende til, når elleve er nok til å regne ut hva som helst?
- **Om grensen.** Vi sa at grensen mellom kode og data ikke finnes i maskinvaren, men håndheves av operativsystemet. Hva slags maskin ville du fått hvis grensen var bygget inn i maskinvaren, og hva ville du mistet?

- **Om abstraksjon.** Assembleren skjuler tallene. Etikettene skjuler adressene. Hva skjuler de *ikke*, som en Python-programmerer aldri trenger å tenke på?
- **Om kontoristen.** Modellen forutsetter et menneske som følger ordrer uten å forstå hva regnestykket gjelder. Hva sier det om hva forståelse er, at arbeidet lot seg overta av en maskin?

3.13 Selv-test

Spørsmålene står i stigende vanskegrad, og fasiten finnes bakerst i boka.

1. Hvor mange postkasser har LMC, og hvorfor akkurat så mange?
2. Del instruksjonen `347` i de to feltene sine, og si hva hvert felt betyr.
3. Hva er forskjellen på innholdet i en postkasse og adressen til den? Gi et eksempel der de to er lette å forveksle.
4. Nevn de fire stegene i hent-utfør-syklusen. I hvilket steg klikkes programtelleren fram?
5. Hva peker programtelleren på mens en instruksjon utføres?
6. Hvorfor er `901` en misvisende instruksjon å lese som «gjør noe med postkasse 01»?
7. Programmet `901 399 901 199 902 000` ligger fra postkasse 00. Brukeren taster inn 40 og deretter 2. Hva står i utkurven, og hva står i postkasse 99 når programmet er ferdig?
8. Skriv om det samme programmet til assembler med etiketter i stedet for talladressen 99.
9. Hvilken opkode er ikke i bruk i LMC?
10. Akkumulatoren inneholder 0. Hopper `BRP 20`? Hopper `BRZ 20`? Begrunn begge svarene.
11. Skriv et LMC-program som leser to tall og skriver ut det største av dem.
12. Skriv et LMC-program som leser et tall og skriver ut det dobbelte. LMC har ingen gangeinstruksjon.
13. Denne løkka skal telle ned fra 5 til 0, men den skriver bare ut ett tall. Finn feilen og rett den.

    ```
    00  LDA 05
    01  OUT
    02  SUB 06
    03  BRZ 01
    04  HLT
    05  5
    06  1
    ```

14. En medstudent sier: «`BRZ` sjekker om instruksjonen over ga null.» Hva er galt med den beskrivelsen, og hvordan ville du formulert den riktig?
15. Hva står det for et tall i postkassen etter linja `svar DAT 7`? Og etter linja `peker DAT svar`, hvis `svar` ligger i postkasse 31?
16. Hvorfor er `DAT` ikke en instruksjon?
17. Et program mangler `HLT`. Første postkasse etter siste instruksjon inneholder tallet 250. Hva gjør maskinen? Hvorfor er det umulig å svare uten å vite hva som ligger i postkasse 50?

18. Se tilbake på figur 3.6 on page 65. Forklar med egne ord hvorfor etiketter løser problemet figuren viser, og hvorfor assembleren må kjøres på nytt hver gang du endrer programmet.

19. I programmet med selvmodifiserende kode hopper `BRA hent` til postkasse 02 og ikke til 00. Hva hadde skjedd hvis det hoppet til 00 i stedet?

20. Skriv et LMC-program som leser inn tall til brukeren taster 0, og som så skriver ut summen av alle tallene.

21. Kapittel 2 sa at et bitmønster ikke betyr noe uten en tolkning. Dette kapitlet sa at et tall i minnet kan være både data og instruksjon. Er dette den samme påstanden? Forklar hva som er felles og hva som er nytt.

22. En maskin kjører et program som har skrevet til en adresse den ikke skulle ha skrevet til. Beskriv tre ulike ting som kan skje videre, avhengig av hva som lå på den adressen.